山东大学
SHANDONG UNIVERSITY

崇新学堂

2024 — 2025 学年第一学期

实验报告

课程名称：Introduction to EECS Lab

实验名称：Design Lab3 - All Carrot, No Stick

学生姓名：胡君安、陈焕斌、黄颢

实验时间：2024 年 10 月 18 日

1 Goals

The overall goal of this lab is to build a robust capability for the robot to drive a route made up of linear path segments. We will do this in three stages:

- Implement a state machine that drives the robot in a straight line to a goal specified in the coordinate system of its odometry.
- Make the robot traverse a sequence of linear segments by cascading a machine that generates target points with the machine that drives to a target.
- Make the robot less of a danger to humanity by adding a 'reflex' that will make the robot stop when its path is blocked, and wait until it is unblocked, then resume its trajectory.

2 Experimental step

Step1: Collect resources

- **moveBrainSkeleton.py:** main brain file, which imports the next two files.
- **ffSkeleton.py:** template for your implementation of a figure-following state machine.
- **dynamicMoveToPointSkeleton.py:** template for your implementation of a behavior that drives the robot to a specified point.
- **testFF.py:** procedure for testing figure follower in idle. Not used when the brain is run.
- **testMove.py:** procedure for testing move to point in idle. Not used when the brain is run.

Step2: Make the composite machine

According to the experimental requirements and figure 1, we created the state machine:

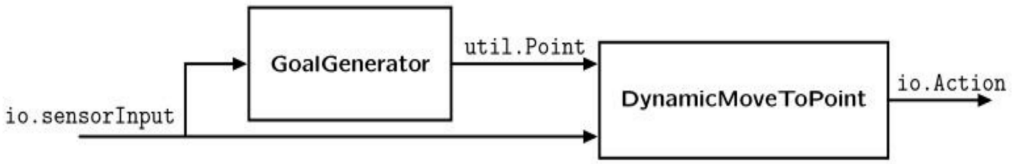

Fig. 1. Composite machine

We edited mySM in moveBrainSkeleton and, based on the schematic diagram of the state machine, combined ffSkeleton with sm Connect WIre in parallel and cascade it with dynamicMoveToPointSkeleton. The specific code implementation is as follows:

```
1      mySM = sm.Cascade(sm.Parallel( ffSkeleton. FollowFigure(secret), sm. Wire()),
2                        ( dynamicMoveToPointSkeleton. DynamicMoveToPoint())
```

The final result is shown in Figure 2:

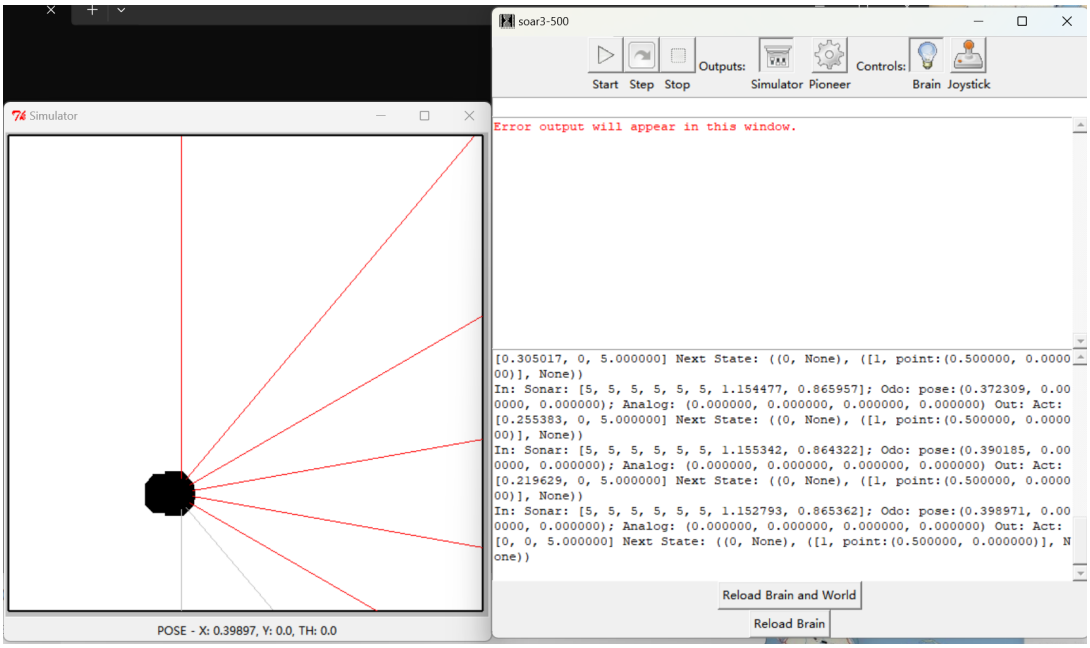

Fig. 2. Final result

Step3: Determine the action

"Determine the action that should be output by the DynamicMoveToPoint state machine for the following input conditions."

Current robot pose	**goalPoint**	**Action**
(0.0, 0.0, 0.0)	(1.0, 0.5)	Rotate left
(0.0, 0.0, $\pi/2$)	(1.0, 0.5)	Rotate right
(0.0, 0.0, $tan^{-1}0.5$)	(1.0, 0.5)	Move forward
(1.0001, 0.4999, 0.0)	(1.0, 0.5)	Stop

Tab. 1. The Action of The Robot

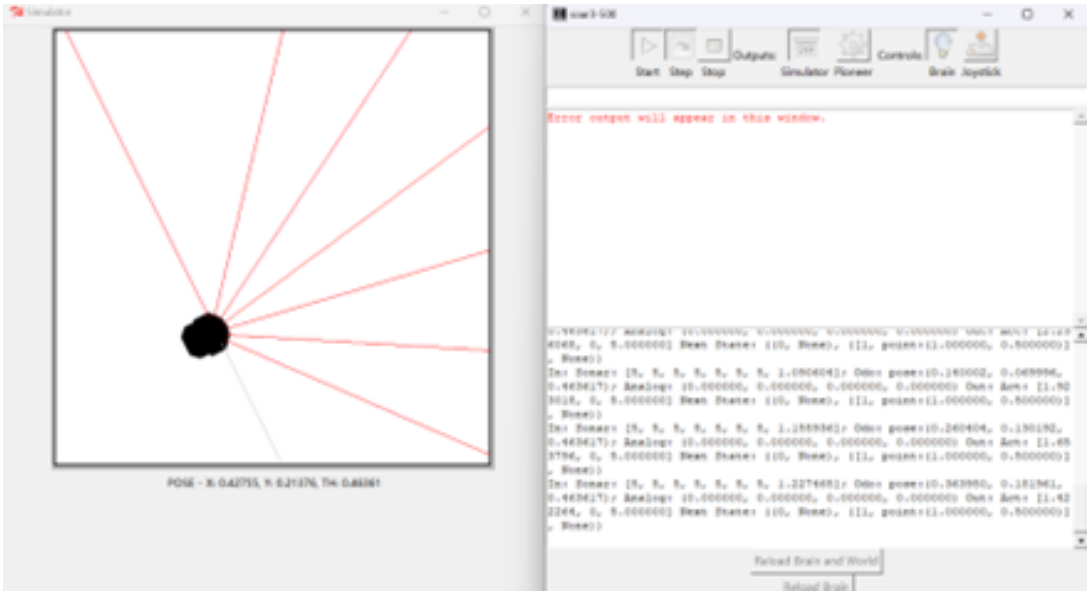

Fig. 3. Reach Point(1.0, 0,5)

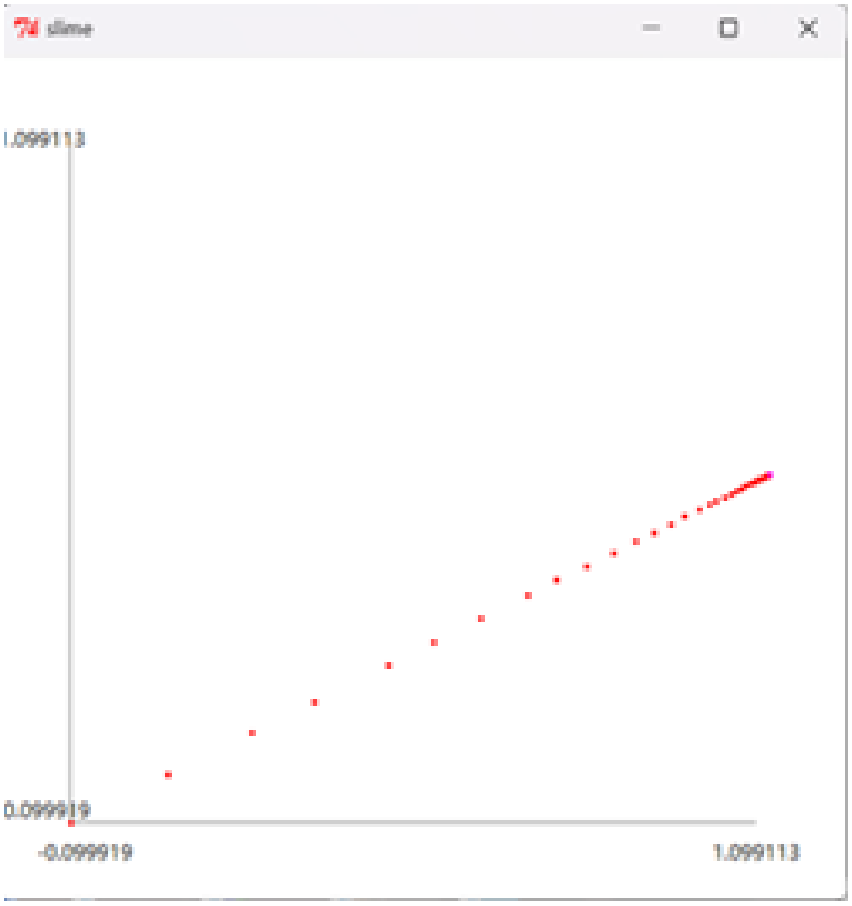

Fig. 4. Slime Figure

Checkoff 1

"Explain your strategy for implementing this behavior and your answers to the questions above to a staff member."

We use position.point() and position.theta to input the current position and angle. We have set three states, namely 0, 1, and 2. When st=0, we set fv to 0 and use NearAngle to obtain the angle difference, and then determine whether this angle difference is less than 0.0001. If the angle difference is too large, loop st=0 to make the robot turn towards goalAngle until the angle difference is less than 0.0001. When the angle difference is less than 0.0001, the robot switches to st=1. When st=1, we set rv to 0 and use point.isNear to obtain the distance difference. Similarly, we determine whether this distance difference is less than 0.001. If the distance difference is too large, correct the distance by looping st=1 until the distance difference is less than 0.001. When the distance difference is less than 0.001, the robot switches to st=2, and the robot stops moving.

We implemented a **proportional control** (P-control) mechanism to guide the robot toward a target point efficiently and smoothly. The proportional control approach was selected for its simplicity and effectiveness in handling continuous, gradual adjustments based on real-time sensor feedback.

Proportional control adjusts the robot' s velocity based on the magnitude of the error, which in this case is the distance or angular deviation from the target point.

The robot rotates until its orientation aligns with the target direction. The rotational velocity is proportional to the difference between the current angle and the target angle. The larger the angular error, the faster the robot turns to correct it.

Once the robot is correctly oriented, it moves forward towards the target point. The forward velocity is proportional to the distance between the current position and the goal. As the robot approaches the goal, the speed decreases proportionally, allowing for a smooth stop.

As a result, in conclusion, our methods' advantages are as follows:

- **Simplicity:** P-control is easy to implement and understand, making it an ideal choice for systems that do not require highly precise control or rapid adaptation to dynamic environments.
- **Smooth Transitions:** By reducing both forward and angular velocity as the robot nears the target, proportional control ensures a smooth, gradual approach, avoiding abrupt stops or turns. This is especially important in real-world navigation, where sudden movements can be unstable or inefficient.
- **Real-time Adjustments:** The system continuously adjusts based on the robot's real-time position and orientation data, allowing it to respond dynamically to changing conditions.
- **Scalable:** The approach can be scaled with appropriate gain tuning (e.g., the constants 10 for angular velocity and 2 for forward velocity) to adapt to different robot dynamics or environmental scenarios.

Step4: Write code

"Write the code for the DynamicMoveToPoint state machine."

```
class  DynamicMoveToPoint(sm.SM):
def __init__(self):
    self.startState = [0, util.Point(0, 0)]

def getNextValues(self, state, inp):
    goal_point = inp[0]
    sensors = inp[1]  # io.SensorInput
    position = sensors.odometry  # util.Pose
    point = position.point()  # util.Point
    angle = position.theta
    goal_angle = point.angleTo(goal_point)
    st1 = state[0]
    st2 = state[1]
    if st2 != goal_point:       #如果当前点不是目标点
        st1 = 0                 #状态1为0
        st2 = goal_point        #状态2为目标点
    if st1 == 0:                #如果状态1为0（即当前点不是目标点）
        fvel = 0                #停下来，准备转向目标点
        if not util.nearAngle(angle, goal_angle, 0.0001):     #如果当前角度和目标角度不
            在可接受范围内
            rvel = -util.fixAnglePlusMinusPi(util.fixAnglePlusMinusPi(angle)
                                    - util.fixAnglePlusMinusPi(goal_angle))*10
                                #角速度的比例增益为10
        else:                   #如果当前角度和目标角度在可接受范围内
            rvel = 0
            st1 = 1
    elif st1 == 1:
        rvel = 0
        if not point.isNear(goal_point, 0.0001):
            fvel = point.distance(goal_point)*2          #直行速度设置为当前点和目标点
                的距离，随着距离的减小，速度逐渐减小，这是一个比例控制+增益，符合实际
        else:
            fvel = 0
            st1 = 2
    else:
        (fvel, rvel) = (0, 0)
    return ([st1, st2], io.Action(fvel, rvel))
```

Check Yourself 1

"Be sure you understand what the answers to the test cases in this file ought to be, and that your code is generating them correctly."

Comment out from soar.io import io and uncomment import lib601.io as io. Then run the testMove.py file to test the DynamicMoveToPoint state machine. The test results are shown in the figure below. The test results are consistent with the expected results, which means that the DynamicMoveToPoint state machine is working properly.

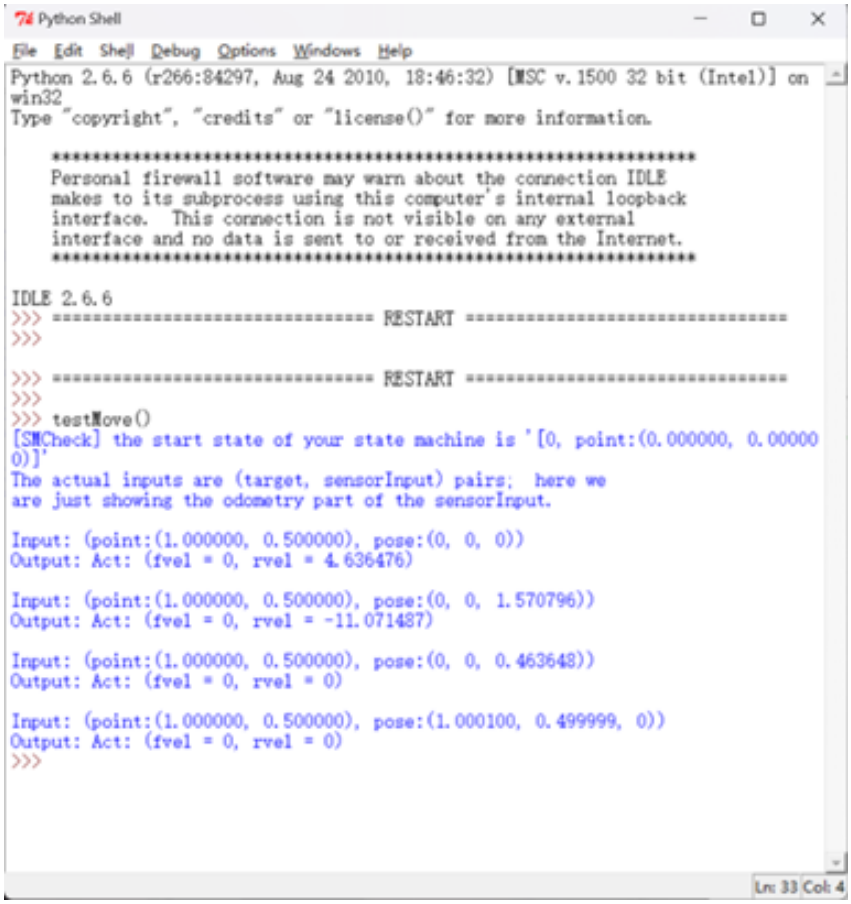

Fig. 5. Test Move

Check Yourself 2

"The robot always starts with odometry reading (0, 0, 0), so it ought to move to somewhere close to the point (1.0, 0.5) and stop."

Comment out import lib601.io as io and uncomment from soar.io import io.

For each state of the robot, the rv or fv of the next state is determined by the angle difference or distance difference, and we provide a parameter for this mechanism to determine its speed rate(for example,' 10' and '2'),just like a cool feedback mechanism. In general, when the robot needs to turn, it turns; when the robot turns to the corresponding angle, it needs to move forward; and when it reaches the finish line, it stops.

Step5: FollowFigure

Current robot pose	State	Target point
(0.0, 0.0, 0.0)	0	(0.5,0.5)
(0.0, 1.0, 0.0)	1	(0.5,0.5)
(0.499, 0.501, 2.0)	2	(0.0,1.0)
(2.0, 3.0, 4.0)	3	(0.0,1.0)

Tab. 2. The state and target point

When we initialize the FollowFigure state machine, we will give it a list of path points that define a sequence of linear segments that the robot should traverse, and that we can determine by distance whether or not it reaches the goal point; it should start by generating the first point in the input sequence as an output, and keep doing so until the robot's actual pose (found as the sensorInput.dometery found) is close to that point; once the robot is close to the goal point, the state machine automatically adds one (switching to generate the next goal point) as an output. Even after the robot approaches the final point, the machine should continue to generate that point as output.

```
1       #The code of FollowFigure
2       class  FollowFigure(sm.SM):
3       def __init__(self, points):
4           self.points = points
5           self.last =  len(points) - 1          #last是最后一个点的索引
6           self.startState = 0
7
8       def getNextValues(self, state, sensors):
9           robot_point = sensors.odometry.point()
10          if state != self.last:           #如果当前状态不是最后一个点
11              if self.points[state].isNear(robot_point, 0.005):        #如果当前点和机器人的距
                    离在可接受范围内
12                  state = state + 1            #导航去下一个点
13          return (state, self.points[state])       #返回下一个状态和下一个点的坐标
```

The first two position points are not reached (0.5,0.5). Therefore ,(0.5,0.5) continues to be output until the third point comes. Then it switches to the next point (0.0,1.0). The fourth point fails to reach the target, so it outputs (0.0,1.0) again.

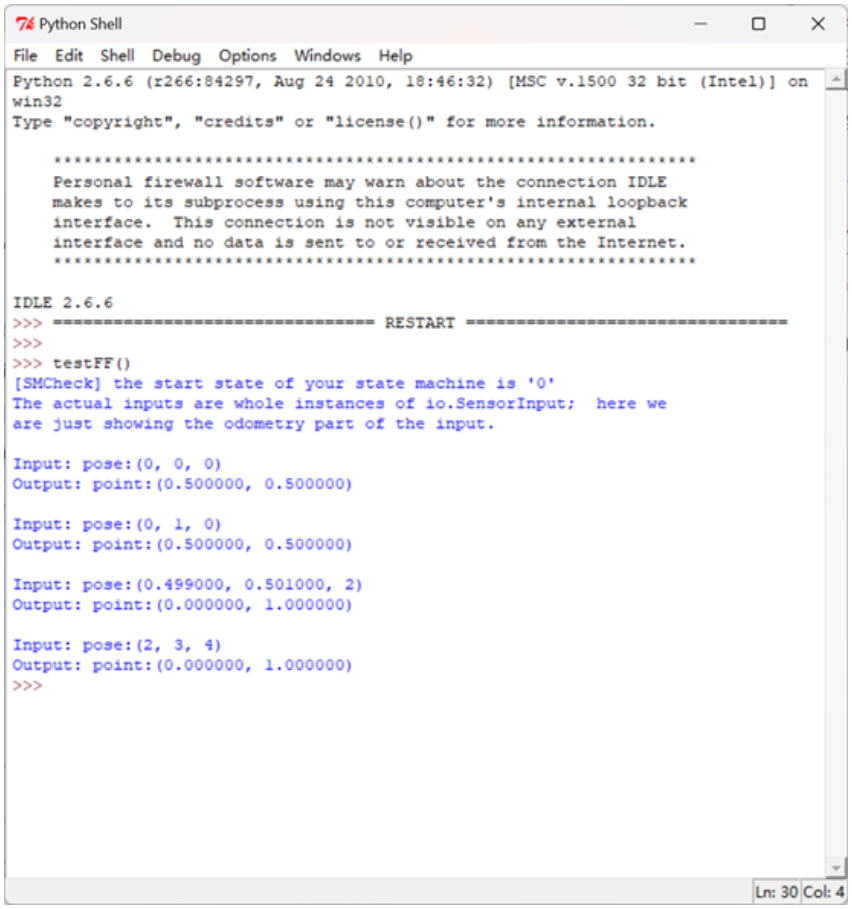

Fig. 6. Final result

Step6: Substitute instance

```
1    #The code of substitution
2    mySM = sm.Cascade(sm.Parallel( ffSkeleton. FollowFigure(secret), sm. Wire()),
         dynamicMoveToPointSkeleton. DynamicMoveToPoint())
```

Checkoff 2

"Show the slime trail resulting from the simulated robot moving in a square or other interesting figure to a staff member. Explain why it has the shape it does. Take a screenshot of the slime trail and save it for your interview."

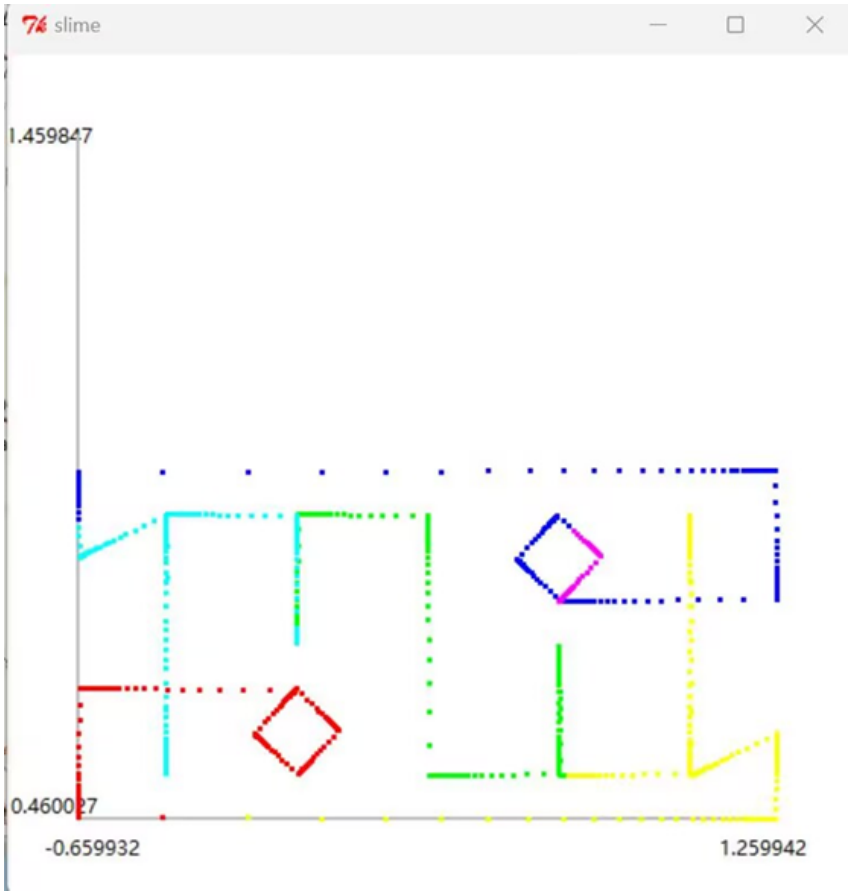

Fig. 7. The slime trail of secretMessage

The slime trail presents a pattern of MIT.

!!!Unwilling to fall behind, we rewrite the code to make a slime trail presenting a pattern of SDU!!!

```
    # trail of SDU
    secret = [util.Point(    5/10.,    0/10.),
              util.Point(    5/10.,    5/10.),
              util.Point(    0/10.,    5/10.),
              util.Point(    0/10.,   10/10.),
              util.Point(    7/10.,   10/10.),
              util.Point(   10/10.,    8/10.),
              util.Point(   10/10.,    2/10.),
              util.Point(    7/10.,    0/10.),
              util.Point(    7/10.,   10/10.),
              util.Point(   12/10.,   10/10.),
              util.Point(   12/10.,    0/10.),
              util.Point(   17/10.,    0/10.),
              util.Point(   17/10.,   10/10.)]
```

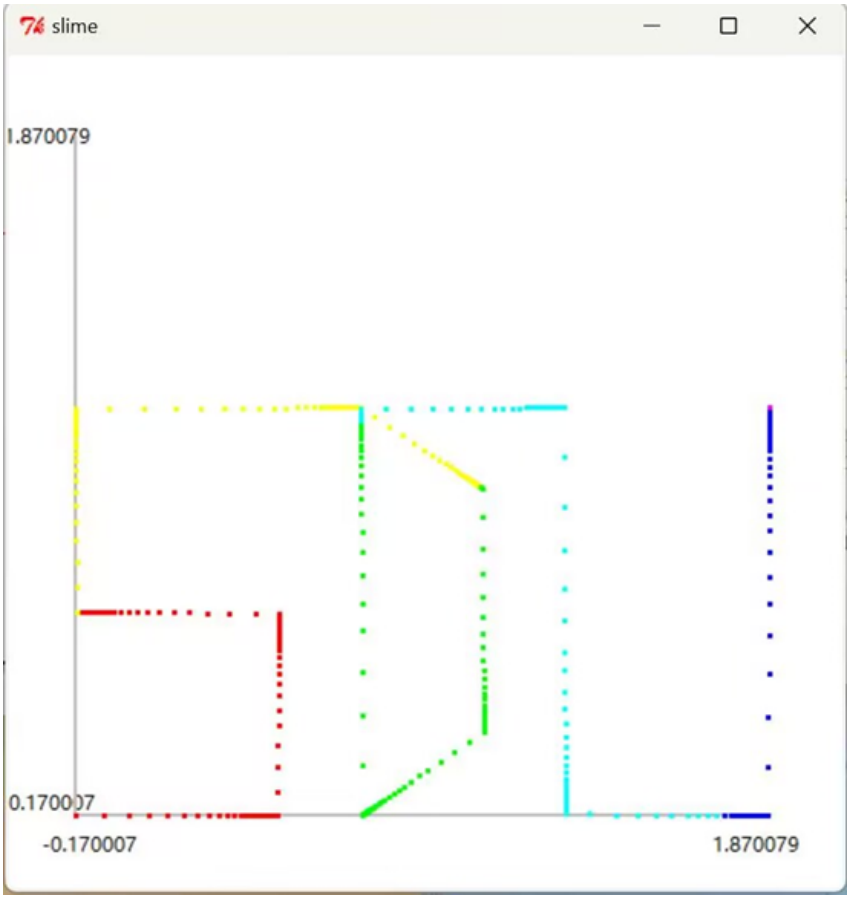

Fig. 8. Trail of SDU

Step7: Avoiding Obstacles

In order to achieve the goal of avoiding, we firstly define a function – cond, which if any of the sonar detects a distance less than 0.3, it return ‘Flase’, otherwise it return ‘Ture’. Then use the sm. Switch state-machine combinator to cascade the state machines to make a robot that stops for obstacles.

```
1    #定义了一个名为 avoid 的函数，实现避障功能
2    #这里的条件是：所有的超声波传感器的读数都大于 0.3，决定是否切换到
         dynamicMoveToPointSkeleton
3    def avoid(inp):
4        sensors = inp[1]
5        for distance in sensors.sonars:
6            if distance < 0.3:
7                return False
8        return True
9
10   # sm.Switch()创建了一个条件切换状态机
11   mySM = sm.Cascade(sm.Parallel( ffSkeleton. FollowFigure(secret), sm.
         Wire()), sm.Switch(avoid,  dynamicMoveToPointSkeleton.
         DynamicMoveToPoint(),sm.Constant(io.Action())))
```

Checkoff 3

"Demonstrate your safe figure-follower to a staff member."

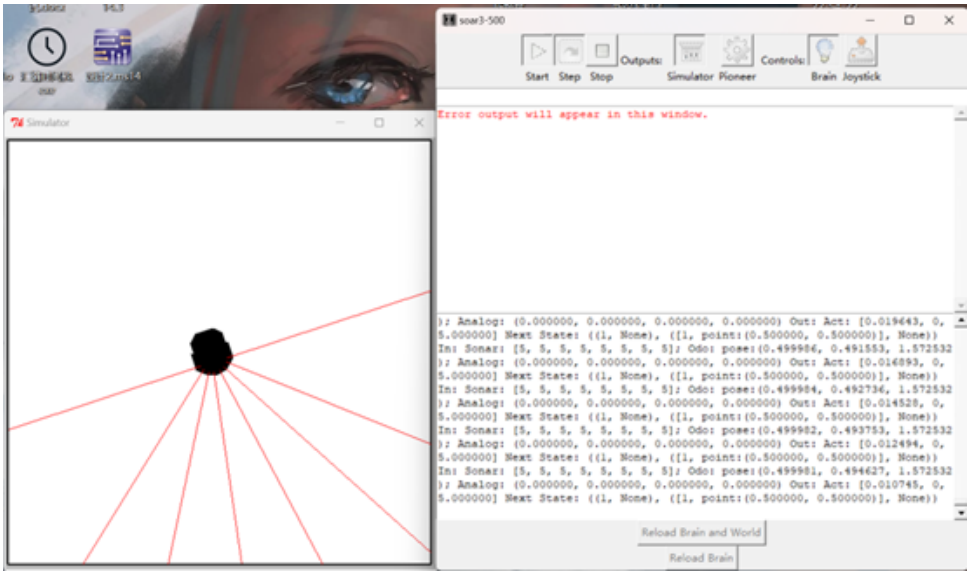

Fig. 9. In the open environment

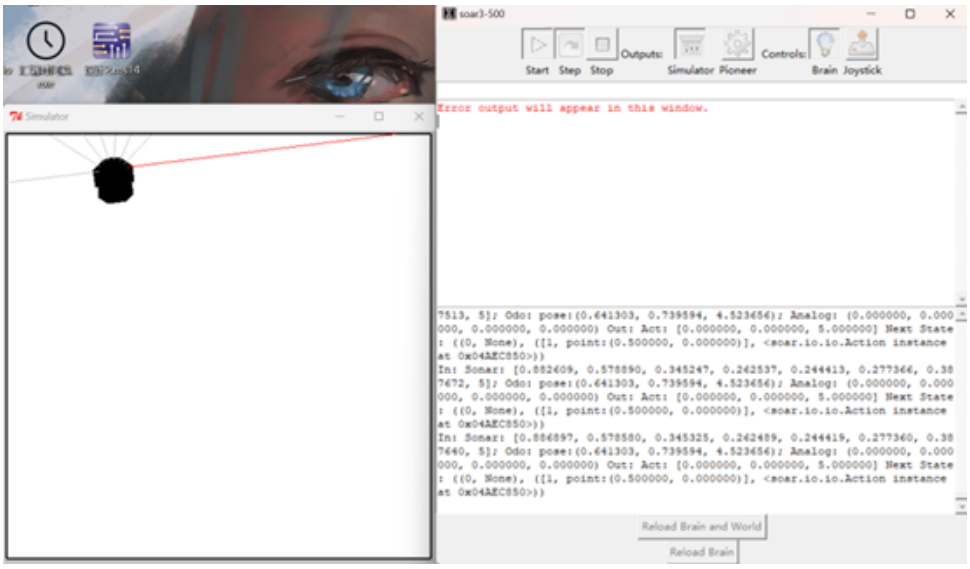

Fig. 10. The Avoiding result

When the robot detects an obstacle, it stops and waits for the obstacle to be removed. After the obstacle is removed, the robot continues to follow the path. The result is shown as expected.

3 Summary

- After studying the cascade of the state machine, we successfully implemented a composite state machine by combining simpler state machines.
- During coding, we mentioned that we ought to handle 'angle arithmetic' carefully, using

helper functions like "fixAnglePlusMinusPi" to avoid discontinuities (ensuring angles near $-\pi$ and π are treated equivalently).

- We used P-control to guide the robot toward a target point efficiently and smoothly. The proportional control approach was selected for its simplicity and effectiveness in handling continuous, gradual adjustments based on real-time sensor feedback.
- When we firstly defined "getNextValue" and ran the "moveBrainSkeleton" in soar, we mentioned that the trail isn't ideal. There's a little off from being perfectly straight. Therefore, we tried to add a gain factor to the proportional control. After a series of attempts, we finally decided to use two coefficients in order to keep straight.
- This experiment truly enhanced our understanding of the link between state machines. We learned how to cascade and parallelize state machines to achieve more complex behaviors. We also learned how to use helper functions to avoid common pitfalls.